

APPLICATION  
FOR  
UNITED STATES PATENT

Entitled

METHOD AND SYSTEM FOR PATTERN MATCHING  
HAVING HOLISTIC TWIG JOINS

Inventor:

Nicolas Bruno  
Nikolaos Koudas  
Divesh Srivastava

Daly, Crowley & Mofford, LLP  
275 Turnpike Street, Suite 101  
Canton, Massachusetts 02021-2310  
Telephone (781) 401-9988  
Facsimile (781) 401-9966

Express Mail Label No. ER476347697US

# METHOD AND SYSTEM FOR PATTERN MATCHING HAVING HOLISTIC TWIG JOINS

## CROSS REFERENCE TO RELATED APPLICATIONS

[0001] The present application claims the benefit of U.S. Provisional Patent Application No. 60/449,648, filed on February 24, 2003, which is incorporated herein by reference.

## STATEMENT REGARDING FEDERALLY SPONSORED RESEARCH

[0002] Not Applicable.

## FIELD OF THE INVENTION

[0003] The present invention relates generally to processing queries in a computer system and, more particularly, to processing computer queries using pattern matching.

## BACKGROUND OF THE INVENTION

[0004] As is known in the art, the eXtensible Markup Language (XML) employs a tree-structured model for representing data. Queries in XML query languages typically specify patterns of selection predicates on multiple elements that have some specified tree structured relationships. For example, the XQuery expression:

book[title = 'XML']//author[fn = 'jane' AND ln = 'doe']

matches author elements that (i) have a child subelement “fn” with content “jane”, (ii) have a child subelement “ln” with content “doe”, and (iii) are descendants of book elements that have a child title subelement with content XML. This expression can be represented as a node-labeled twig (or small tree) pattern with elements and string values as node labels.

[0005] Finding all occurrences of a twig pattern in a database is a core operation in XML query processing, both in relational implementations of XML databases, and in native XML databases. Known processing techniques typically decompose the twig pattern into

a set of binary (parent-child and ancestor-descendant) relationships between pairs of nodes, e.g., the parent-child relationships (book, title) and (author, fn), and the ancestor-descendant relationship (book, author). The query twig pattern can then be matched by (i) matching each of the binary structural relationships against the XML database, and (ii) “stitching” together these basic matches.

[0006] In one known attempt at solving the first sub-problem of matching binary structural relationships, Zhang et al., “On Supporting Containment Queries in Relational Database Management Systems,” Proceedings of ACM SIGMOD, 2001, (hereafter “Zhang”), proposed a variation of the traditional merge join algorithm, the multi-predicate merge join (MPMGJN) algorithm, based on the (DocId, LeftPos RightPos, LevelNum) representation of positions of XML elements and string values. Zhang’s results showed that the MPMGJN algorithm could outperform standard RDBMS join algorithms by more than an order of magnitude. Zhang is incorporated herein by reference.

[0007] A further sub-problem of stitching together the basic matches obtained using binary “structural” joins requires identifying a ‘good’ join ordering in a computational cost-based manner taking selectivities and intermediate result size estimates into account. A basic limitation of this traditional approach for matching query twig patterns is that intermediate result sizes can get quite large, even when the input and final result sizes are more manageable.

[0008] It would, therefore, be desirable to overcome the aforesaid and other disadvantages.

## SUMMARY OF THE INVENTION

[0009] The present invention provides optimal query pattern matching. In one embodiment, each node in query twig pattern is associated with a respective stream containing positional representations of the database nodes that match the node predicate

at the twig pattern node. The nodes in the streams are sorted using one or more attribute values, such as document ID and left position. Each query node is associated with a respective stack and each data node in the stacks includes a pair: a positional representation of node from the stream, and a pointer to a node in a stack containing the parent node for the node. During the computations, the nodes in the stacks from bottom to top are guaranteed to lie on a root-to-leaf path in the database and, the set of stacks contain a compact encoding of partial and total answers to the query twig pattern.

#### BRIEF DESCRIPTION OF THE DRAWINGS

[0010] The invention will be more fully understood from the following detailed description taken in conjunction with the accompanying drawings, in which:

[0011] FIG. 1 is a tree representation of an exemplary XML document that can be processed in accordance with the present invention;

[0012] FIGs. 2A-2B are exemplary query twig patterns corresponding to queries that can be processed in accordance with the present invention;

[0013] FIGs. 3A-3D are pictorial representations of compact encoding of answers using stacks;

[0014] FIG. 4 is a textual representation of the pathstack algorithm in accordance with the present invention;

[0015] FIG. 4A is a flow diagram showing an exemplary implementation of the pathstack algorithm of FIG. 4;

[0016] FIG. 5 is a textual representation of the showsolutions procedure in accordance with the present invention;

[0017] FIG. 6 is a pictorial representation of various cases for the pathstack and twigstack algorithms in accordance with the present invention;

[0018] FIG. 7 is a textual representation of the pathMPMJ algorithm in accordance with the present invention;

[0019] FIG. 8 is a textual representation of the twigstack algorithm in accordance with the present invention;

[0020] FIG. 8A is a flow diagram showing an exemplary sequence of steps for implementing the algorithm of FIG. 8;

[0021] FIG. 9 is a textual representation of the twigstackXB algorithm in accordance with the present invention;

[0022] FIG. 9A is a flow diagram showing an exemplary sequence of steps for implementing the algorithm of FIG. 9;

[0023] FIG. 10 is a graphical depiction of holistic and binary joins for path queries;

[0024] FIG. 11 is a graphical depiction of pathMPMJ versus pathMPMJNative;

[0025] FIG. 12A is a graphical depiction of execution time versus pathlength for pathstack and pathMPMJ;

[0026] FIG. 12B is a graphical depiction of the number of elements read versus pathlength for pathstack and pathMPMJ;

[0027] FIG. 13A is a graphical depiction of execution time versus an unfolded DBLP data set for pathstack and pathMPMJ;

[0028] FIG. 13B is a graphical depiction of the number of elements read versus an unfolded DBLP data set for pathstack and pathMPMJ;

[0029] FIGs. 14A-14C are pictorial representations of twig queries processed in accordance with the present invention;

[0030] FIGs. 15A-15C are graphical representations of performance characteristics for pathstack and twigstack for first and second twig queries;

[0031] FIGs. 16A-16C are graphical representations of performance characteristics for pathstack and twigstack for a parent-child twig query;

[0032] FIGs. 17A-17B are graphical representations of performance characteristics for pathstack and twigstack on a data set; and

[0033] FIG. 18A-18C are graphical representations of the number of elements read versus node capacity using XB trees.

## DETAILED DESCRIPTION OF THE INVENTION

[0034] The present invention provides a holistic twig join algorithm, “TwigStack,” for matching an XML query twig pattern. The Twigstack algorithm uses a chain of linked stacks to compactly represent partial results to root-to-leaf query paths, which are then composed to obtain matches for the twig pattern. When the twig pattern uses only ancestor-descendant relationships between elements, TwigStack is I/O and CPU optimal among sequential algorithms that read the entire input: it is linear in the sum of sizes of the input lists and the final result list, and independent of the sizes of intermediate results. In another aspect of the invention, a modification of so-called B-trees can be used, along with the TwigStack algorithm, to match query twig patterns in sub-linear time.

[0035] The inventive holistic twig join approach for matching XML query twig patterns creates relatively small intermediate results. Processing uses the (DocId, LeftPos : RightPos, LevelNum) representation of positions of XML elements and string values that succinctly capture structural relationships between nodes in the XML database. The inventive Twigstack algorithm can also use a chain of linked stacks to compactly represent partial results to individual query root-to-leaf paths, which are then composed to obtain matches to the query twig pattern. Since a relatively large amount of XML data is expected to be stored in relational database management systems (RDBMS), such as from Oracle, IBM and Microsoft, it will be appreciated that RDBMS systems will benefit from the inventive query processing using holistic twig joins for efficient XML query processing. It is understood that the invention is also applicable to native XML query engines, since holistic twig joins are an efficient, set-at-a-time strategy for matching XML query patterns, in contrast to the node-at-a-time approach of using tree traversals.

[0036] Before describing the invention in detail, some background information is presented below. An XML database is a forest of rooted, ordered, labeled trees, with each node corresponding to an element or a value and the edges representing (direct) element-subelement or element-value relationships. Node labels include a set of (attribute, value) pairs, which suffices to model tags, IDs, IDREFs, etc. The ordering of sibling nodes implicitly defines a total order on the nodes in a tree, obtained by a preorder traversal of the tree nodes.

[0037] FIG. 1 shows a tree representation of an exemplary XML document. Queries in XML query languages, such as XQuery, Quilt, and XML-QL, make use of (node labeled) twig patterns for matching relevant portions of data in the XML database. The twig pattern node labels include element tags, attribute-value comparisons, and string values, and the query twig pattern edges are either parent-child edges (depicted using a single line) or ancestor-descendant edges (depicted using a double line). For example, the XQuery expression:

book[title = 'XML' AND year = '2000']

which matches book elements that (i) have a child title subelement with content “XML”, and (ii) have a child year subelement with content “2000”, can be represented as the twig pattern in FIG. 2A. It is understood that only parent-child edges are used in this case. Similarly, the previously described XQuery expression can be represented as the twig pattern in FIG. 2B. Note that an ancestor-descendant edge is used between the book element and the author element.

[0038] In general, at each node in the query twig pattern, there is a node predicate on the attributes (e.g., tag, content) of the node in question. It is understood that for the present invention, what is permitted in this predicate is not material. Similarly, the physical representation of the nodes in the XML database is not relevant to the results set forth below. It suffices to say that there should be efficient access mechanisms (such as index structures) to identify the nodes in the XML database that satisfy any given node predicate  $q$ , and return a stream of matches  $T_q$ .

[0039] Given a query twig pattern  $Q$  and an XML database  $D$ , a match of  $Q$  in  $D$  is identified by a mapping from nodes in  $Q$  to nodes in  $D$ , such that: (i) query node predicates are satisfied by the corresponding database nodes (the images under the mapping), and (ii) the structural (parent-child and ancestor-descendant) relationships between query nodes are satisfied by the corresponding database nodes. The answer to query  $Q$  with  $n$  nodes can be represented as  $n$ -ary relation where each tuple  $(d_1, \dots, d_n)$  includes the database nodes that identify a distinct match of query twig pattern  $Q$  in database  $D$ .

[0040] Finding matches of a query twig pattern in an XML database is a core operation in XML query processing, both in relational implementations of XML databases, and in native XML databases. Consider the twig pattern matching problem: Given a query twig pattern  $Q$ , and an XML database  $D$  that has index structures to identify database nodes that satisfy each of  $Q$ 's node predicates, compute the answer to  $Q$  on  $D$ .

[0041] Consider, for example, the query twig pattern in FIG. 2A, and the database tree in FIG. 1. This query twig pattern has one match in the data tree that maps the nodes in the query to the root of the data and its first and third subtrees. One factor in providing an efficient, uniform mechanism for set-at-a-time (join-based) matching of query twig patterns is a positional representation of occurrences of XML elements and string values in the XML database, which extends the classic inverted index data structure in information retrieval.

[0042] The position of a string occurrence in the XML database can be represented as a 3-tuple (DocId, LeftPos, LevelNum), and analogously, the position of an element occurrence as a 3-tuple (DocId, LeftPos : RightPos, LevelNum), where (i) DocId is the identifier of the document; (ii) LeftPos and RightPos can be generated by counting word numbers from the beginning of the document DocId until the start and the end of the element, respectively; and (iii) LevelNum is the nesting depth of the element (or string value) in the document. FIG. 1 shows 3-tuples associated with some tree nodes, based on this representation. Note that the DocId for all nodes is chosen to be one.

[0043] Structural relationships between tree nodes whose positions are recorded in this fashion can be determined easily: (i) ancestor-descendant: a tree node  $n_2$  whose position in the XML database is encoded as  $(D_2, L_2 : R_2, N_2)$  is a descendant of a tree node  $n_1$  whose position is encoded as  $(D_1, L_1 : R_1, N_1)$  if, and only if (iff),  $D_1 = D_2$ ,  $L_1 < L_2$ , and  $R_2 < R_1$  (It is understood that for leaf strings, the RightPos value is the same as the LeftPos value.); and (ii) parent-child: a tree node  $n_2$  whose position in the XML database is encoded as  $(D_2, L_2 : R_2, N_2)$  is a child of a tree node  $n_1$  whose position is encoded as  $(D_1, L_1 : R_1, N_1)$  iff  $D_1 = D_2$ ,  $L_1 < L_2$ ,  $R_2 < R_1$ , and  $N_1 + 1 = N_2$ . For example, in FIG. 1, the author node with position (1,6 : 20, 3) is a descendant of the book node with position (1, 1 : 150, 1), and the string “jane” with position (1,8,5) is a child of the author node with position (1, 7: 9,4).

[0044] It can be noted that in this representation of node positions in the XML data tree, checking an ancestor-descendant relationship is as simple as checking a parent-child relationship (one can check for an ancestor-descendant structural relationship without knowledge of the intermediate nodes on the path). Also, this representation of positions of nodes allows for checking order (e.g., node  $n_2$  follows node  $n_1$ ) and structural proximity (e.g., node  $n_2$  is a descendant within three levels of  $n_1$ ) relationships.

[0045] Let  $q$  (with or without subscripts) denote twig patterns, as well as (interchangeably) the root node of the twig pattern. In the inventive algorithms, use is made of the following twig node operations:  $\text{isLeaf}: \text{Node} \rightarrow \text{Bool}$ ,  $\text{isRoot}: \text{Node} \rightarrow \text{Bool}$ ,  $\text{parent}: \text{Node} \rightarrow \text{Node}$ ,  $\text{children}: \text{Node} \rightarrow \{\text{Node}\}$ , and  $\text{subtreeNodes}: \text{Node} \rightarrow \{\text{Node}\}$ , where  $\text{isLeaf}$  checks if the operand is a leaf node,  $\text{isRoot}$  checks if the operand is a root node,  $\text{parent}$ : returns the parent of the operand,  $\text{children}$ : returns the set of children of the operand,  $\text{subtreeNodes}$  returns the set of descendants of the operand. Path queries have only one child per node, otherwise the function  $\text{children}(q)$  returns the set of children nodes of  $q$ . The result of operation  $\text{subtreeNodes}(q)$  is the node  $q$  and all its descendants.

[0046] Associated with each node  $q$  in a query twig pattern there is a stream  $T_q$ . The stream contains the positional representations of the database nodes that match the node predicate at the twig pattern node  $q$  (possibly obtained using an efficient access mechanism, such as an index structure). The nodes in the stream are sorted by their  $(\text{DocId}, \text{LeftPos})$  values. The operations over streams are:  $\text{eof}$ ,  $\text{advance}$ ,  $\text{next}$ ,  $\text{nextL}$ , and  $\text{nextR}$ . The last two operations return the  $\text{LeftPos}$  and  $\text{RightPos}$  coordinates in the positional representation of the next element in the stream, respectively.

[0047] In the inventive stack-based algorithms,  $\text{PathStack}$  and  $\text{TwigStack}$ , each query node  $q$  is also associated with a stack  $S_q$ . Each data node in the stack includes a pair: (positional representation of a node from  $T_q$ , pointer to a node in  $S_{\text{parent}(q)}$ ). The operations over stacks are:  $\text{empty}$ ,  $\text{pop}$ ,  $\text{push}$ ,  $\text{topL}$ , and  $\text{topR}$ . The last two operations

return the LeftPos and RightPos coordinates in the positional representation of the top element in the stack, respectively. At every point during the computation, (i) the nodes in stack  $S_q$  (from bottom to top) are guaranteed to lie on a root-to-leaf path in the XML database, and (ii) the set of stacks contains a compact encoding of partial and total answers to the query twig pattern, which can represent in linear space a potentially exponential (in the number of query nodes) number of answers to the query twig pattern, as illustrated below.

#### EXAMPLE

[0048] FIGs. 3A-D illustrate the stack encoding of answers to a path query for a sample data set. The answer  $[A_2, B_2, C_1]$  is encoded since  $C_1$  points to  $B_2$ , and  $B_2$  points to  $A_2$ . Since  $A_1$  is below  $A_2$  on the stack  $S_A$ ,  $[A_1, B_2, C_1]$  is also an answer. Finally, since  $B_1$  is below  $B_2$  on the stack  $S_B$ , and  $B_1$  points to  $A_1$ ,  $[A_1, B_1, C_1]$  is also an answer. Note that  $[A_2, B_1, C_1]$  is not an answer, since  $A_2$  is above the node ( $A_1$ ) on stack  $S_A$  to which  $B_1$  points. The relatively compact stack encoding is used in the inventive PathStack and TwigStack algorithms.

[0049] Algorithm PathStack, which computes answers to a query path pattern, is presented in FIG. 4 for the case when the streams contain nodes from a single XML document. When the streams contain nodes from multiple XML documents, the algorithm is easily extended to test equality of DocId before manipulating the nodes in the streams and stacks.

[0050] One feature of Algorithm PathStack is to repeatedly construct (compact) stack encodings of partial and total answers to the query path pattern, by iterating through the stream nodes in sorted order of their LeftPos values; thus, the query path pattern nodes will be matched from the query root down to the query leaf. Line 2, in Algorithm PathStack, identifies the stream containing the next node to be processed. Lines 3-5 remove partial answers from the stacks that cannot be extended to total answers, given knowledge of the next stream node to be processed. Line 6 augments the partial answers

encoded in the stacks with the new stream node. Whenever a node is pushed on the stack

$S_{q_{\min}}$ , where  $q_{\min}$ , is the leaf node of the query path, the stacks contain an encoding of total answers to the query path, and Algorithm showSolutions is invoked by Algorithm PathStack (lines 7—9) to “output” these answers.

[0051] One way for Algorithm showSolutions to output query path answers encoded in the stacks is as n-tuples that are sorted in leaf-to-root order of the query path. This will ensure that, over the sequence of invocations of Algorithm showSolutions by Algorithm PathStack, the answers to the query path are also computed in leaf-to-root order.

[0052] FIG. 4A shows an exemplary sequence of steps for implementing the pathstack algorithm in accordance with the present invention. In step 100, it is determined whether the set of streams  $q$  in the subtree rooted at node  $q$  is not empty (line 01 in FIG. 4). That is, one tests if every stream associated with a node in the subtree rooted at  $q$  is empty. If so, the process terminates. If not, in step 102 the node having a minimum ordering value  $q_i$  is retrieved for processing (function getMinsource, line 02). Partial answers that cannot extend to full answers to the query are removed in step 104 (lines 03-05). In step 106, the partial answers are augmented with the next element in the stream  $q_i$  (line 06). In step 108, it is determined whether the node  $q_i$  is a leaf node (line 07). If not, processing continues in step 100. If so, in step 110 solutions are produced (line 08, see showSolutions algorithm in FIG. 5 below).

[0053] FIG. 5 shows an exemplary showSolutions procedure for the case when only ancestor-descendant edges are present in the query path. When parent-child edges are present in the query path, the LevelNum information should be taken into account. PathStack does not need to change, but it should be ensured that each time showSolutions is invoked, it does not output incorrect tuples, in addition to avoiding unnecessary work. This can be achieved by modifying the recursive call (lines 6-7) to check for parent-child edges, in which case only a single recursive call

(showSolutions(SN- 1, S[SN].index[SN].pointer\_to\_the\_parent\_stack))

needs to be invoked, after verifying that the LevelNum of the two nodes differ by one. Looping through all nodes in the stack  $S[SN - 1]$  would still be correct, but it would do more work than is strictly necessary.

[0054] If it is desired that the final answers to the query path be presented in sorted root-to-leaf order (as opposed to sorted leaf-to-root order), it is easy to see that it does not suffice that each invocation of algorithm showSolutions outputs answers encoded in the stack in the root-to-leaf order. As will be appreciated by one of ordinary skill in the art, to produce answers in the sorted root-to-leaf order, the answers should be “blocked,” and their output delayed until it is certain that no answer prior to them in the sort, order can be computed.

#### EXAMPLE

[0055] Consider the leftmost path, book-title-XML, in each of the query twigs of FIG. 2. If conventional binary structural join algorithms are used, one would first need to compute matches to one of the parent-child structural relationships: book-title, or title-XML. Since every book has a title, this binary join would produce a lot of matches against an XML books database, even when there are only a few books whose title is XML. If, instead, matches to title-XML are computed, there would also be matching of pairs under chapter elements, as in the XML data tree of FIG. 1, which do not extend to total answers in the query path pattern.

[0056] Using the inventive Algorithm PathStack, partial answers are compactly represented in the stacks, and not output. Using the XML data tree of FIG. 1, only one total answer identified by the mapping [book  $\rightarrow$  (1, 1 : 150,1), title  $\rightarrow$  (1, 2 : 4,2), XML  $\rightarrow$  (1, 3,3)], is encoded in the stacks.

[0057] From FIG. 6 it can be seen that if node Y is fixed, the sequence of cases between node Y and nodes X on increasing order of LeftPos (L) is: (1|2)\*3\*4\*. Cases 1 and 2 are

interleaved, then all nodes in Case 3, before any node in Case 4, and finally all nodes in Case 4.

[0058] Suppose that for an arbitrary node  $q$  in the path pattern query, one has function  $\text{getMinSource}(q) = q_N$ . Also, suppose that  $t_{q_N}$  is the next element in  $q_N$ 's stream. Then, after  $t_{q_N}$  is pushed on to stack  $S_{q_N}$ , the chain of stacks from  $S_{q_N}$  to  $S_q$  verifies that their labels are included in the chain of nodes in the XML data tree  $t_{q_N}$  to the root.

[0059] For each node  $t_{q_{\min}}$  pushed on stack  $S_{q_{\min}}$ , it is relatively easy to see that the above property, along with the iterative nature of Algorithm `showSolutions`, ensures that, all answers in which  $t_{q_{\min}}$  is a match for query node  $q_{\min}$  will be output. This leads to the conclusion that, given a query path pattern  $q$  and an XML database  $D$ , Algorithm `PathStack` correctly returns all answers for  $q$  on  $D$ .

[0060] Optimality of the inventive `PathStack` algorithm is now discussed. Given an XML query path of length  $n$ , `PathStack` takes  $n$  input lists of tree nodes sorted by (`DocId`, `LeftPos`), and computes an output sorted list of  $n$ -tuples that match the query path. It is straightforward to see that, excluding the invocations to `showSolutions`, the I/O and CPU costs of `PathStack` are linear in the sum of sizes of the  $n$  input lists. Since the "cost" of `showSolutions` is proportional to the size of the output list, the optimality result can be expressed as follows: given a query path pattern  $q$  with  $n$  nodes, an XML database, Algorithm `PathStack` has worst-case I/O and CPU time complexities linear in the sum of sizes of the  $n$  input lists and the output list. Further, the worst-case space complexity of Algorithm `PathStack` is the minimum of (i) the sum of sizes of the  $n$  input lists, and (ii) the maximum length of a root-to-leaf path in  $D$ . It should be noted that the worst-case time complexity of Algorithm `PathStack` is independent of the sizes of any intermediate results.

[0061] A straightforward generalization of the known MPMGJN algorithm for path queries proceeds one stream at a time to get all solutions. Consider the path query

$q_1//q_2//q_3$ . The basic steps are as follows: Get the first (next) element from the stream  $T_{q_1}$  and generate all solutions that use that particular element from  $T_{q_1}$ . Then, advance  $T_{q_1}$  and backtrack  $T_{q_2}$  and  $T_{q_3}$ , accordingly (i.e., to the earliest position that might lead to a solution). This procedure is repeated until  $T_{q_1}$  is empty. The generate all solutions step recursively starts with the first marked element in  $T_{q_2}$ , gets all solutions that use that element (and the calling element in  $T_{q_1}$ ), then advances the stream  $T_{q_2}$  until there are no more solutions within the current, element in  $T_{q_2}$ , and so on. This algorithm can be referred to as PathMPMJNaive.

[0062] It can be seen that maintaining only one mark per stream (for backtracking purposes) is relatively inefficient, since all marks need to point to the earliest segment that can match the current element in  $T_{q_1}$  (time stream of the root node). An alternative strategy is to use a stack of marks, as shown in Algorithm PathMPMJ of FIG. 7. In this optimized generalization of MPMGJN, each query node will not have a single mark in the stream, but “k” marks, where k is the number of its ancestors in the query. Each mark points to an earlier position in the stream, and for query node q, the i’t h mark is the first point in  $T_q$  such that the element in  $T_q$  starts after the current. element in the stream of q’s i’t h ancestor. Thus, given a query path pattern q and an XML database D, Algorithm PathMPMJ correctly returns all answers for q on D.

[0063] In another aspect of the invention, twig join algorithms are provided. A straightforward way of computing answers to a query twig pattern is to decompose the twig into multiple root-to-leaf path patterns, use PathStack to identify solutions to each individual path, and then merge-join these solutions to compute the answers to the query. This approach, which was evaluated as described below, faces the same fundamental problem as techniques based on binary structural joins, towards a holistic solution: many intermediate results may not be part of any final answer, as illustrated below.

#### EXAMPLE

[0064] Consider the query sub-twig rooted at the author node of the twig pattern in FIG.

2B. Against the XML database in FIG. 1, the two paths of this query: author-fn-jane, and author-ln-doe, have two solutions each, but the query twig pattern has only one solution.

[0065] In general, if the query (root-to-leaf) paths have many solutions that do not contribute to the final answers, using PathStack (as a sub-routine) is suboptimal, in that the over-all computation cost for a twig pattern is proportional not just to the sizes of the input and the final output, but also to the sizes of intermediate results. In one embodiment, this suboptimality is overcome using Algorithm TwigStack.

[0066] Algorithm TwigStack, which computes answers to a query twig pattern, is presented in FIG. 8, for the case when the streams contain nodes from a single XML document. As with Algorithm PathStack, when the streams contain nodes from multiple XML documents, the algorithm is readily extendable to test equality of DocId before manipulating the nodes in the streams and on the stacks.

[0067] In one embodiment, Algorithm TwigStack operates in two phases. In the first phase (lines 1-11) shown in FIG. 8, some (but not all) solutions to individual query root-to-leaf paths are computed. In the second phase (line 12), these solutions are merge-joined to compute the answers to the query twig pattern.

[0068] FIG. 8A shows an exemplary sequence of steps for algorithm TwigStack of FIG. 8. In step 200, it is determined whether the set of streams  $q$  is not empty (one checks all the streams below  $q$ ) (line 01 in code shown in FIG. 8). If so, assuming pass solutions have been processed as described below, in step 202 path solutions are merged (line 12). If not, in step 204 the next node  $qact$  is retrieved after confirming that the node has a descendant in each of the streams involved in the query and recursively checking that the descendants satisfy this property (line 02, function getNext( $q$ )). In step 206, it is determined whether the node  $qact$  is not a root node. If so, in step 208, before proceeding to step 210, the stack is cleaned of contained partial solutions involving the parent of node  $qact$ . In step 210, the no branch from step 206, it is determined whether the node

qact is a root node or the stack of qact's parent is not empty (line 05). If not, in step 212 the stream containing node qact is advanced (line 11) and processing continues in step 200. If so, step 214 the stack involving the node qact is cleaned and in step 216 the node qact is added to the stack extending partial solutions. In step 218, it is determined whether the node qact is a leaf node. If not, processing continues in step 212. If so, in step a solution is generated with blocking and processing continues in step 200.

[0069] One difference between PathStack and the first phase of TwigStack is that before a node  $h_q$  from the stream  $T_q$  is pushed on its stack  $S_q$ , TwigStack (via its call to getNext) ensures that: (i) node  $h_q$  has a descendant  $h_{q_i}$  in each of the streams  $T_{q_i}$ , for  $q_i \in \text{children}(q)$ , and (ii) each of the nodes  $h_{q_i}$  recursively satisfies the first property.

Algorithm PathStack does not satisfy this property (and it does not need to do so to ensure (asymptotic) optimality for query path patterns). Thus, when the query twig pattern has only ancestor-descendant edges, each solution to each individual query root-to-leaf path is guaranteed to be merge-joinable with at least one solution to each of the other root-to-leaf paths. This ensures that no intermediate solution is larger than the final answer to the query twig pattern.

[0070] The second merge-join phase of Algorithm TwigStack is linear in the sum of its input (the solutions to individual root-to-leaf paths) and output (the answer to the query twig pattern) sizes, only when the inputs are in sorted order of the common prefixes of the different query root-to-leaf paths. This requires that the solutions to individual query paths be output in root-to-leaf order as well, which necessitates blocking; showSolutions (shown in FIG. 5), which outputs solutions in sorted leaf-to-root order, cannot be used.

## EXAMPLE 2

[0071] Consider again the query of Example 1, which is the sub-twig rooted at the author node of the twig pattern in FIG. 2B, and the XML database tree in FIG. 1. Before Algorithm TwigStack pushes an author node on the stack  $S_{\text{author}}$ , it ensures this author node has: (i) a descendant fn node in the stream  $T_{\text{fn}}$  (which in turn has a descendant jane

node in  $T_{jane}$ ), and (ii) a descendant  $ln$  node in the stream  $T_{ln}$  (which in turn has a descendant  $doe$  node in  $T_{doe}$ ). Thus, only one of the three author nodes (corresponding to the third author) from the XML data tree in FIG. 1 is pushed on the stacks. Subsequent steps ensure that only one solution to each of the two paths of this query:  $author-fn-jane$ , and  $author-ln-doe$ , is computed. Finally, the merge-join phase computes the desired answer.

[0072] Consider a twig query  $Q$ . For each node  $q \in subtreeNodes(Q)$  one can define the head of  $q$ , denoted  $h_q$ , as the first element in  $T_q$  that participates in a solution for the sub-query rooted at  $q$ . One can say that a node  $q$  has a minimal descendant extension if there is a solution for the sub-query rooted at  $q$  composed entirely of the head elements of  $subtreeNodes(q)$ .

[0073] Suppose that for an arbitrary node  $q$  in the twig query tree there is that  $getNext(q) = q_N$ . Then the following properties hold:

- $q_N$  has a minimal descendant extension.
- For each node  $q' \in subtreeNodes(q_N)$ , the first element in  $T_{q'}$  is  $h_{q'}$ .
- Either (a)  $q = q_N$  or (b)  $parent(q_N)$  does not have a minimal right extension because of  $q_N$  (and possibly other nodes). In other words, the solution rooted at  $p = parent(q_N)$  that uses  $h_p$  does not use  $h_q$  for node  $q$  but some other element whose  $L$  component is larger than that of  $h_q$ .

[0074] Thus, when some node  $q_N$  is returned by  $getNext$ ,  $h_{q_N}$  is guaranteed to have a descendant extension in  $subtreeNodes(q_N)$ . It can also be seen that any element in the ancestors of  $q_N$  that uses  $h_{q_N}$  in a descendant extension was returned by  $getNext$  before  $h_{q_N}$ . Therefore one can maintain, for each node  $q$  in the query, the elements that are part of a solution involving other elements in the streams of  $subtreeNodes(q)$ . Then, each time that  $q_N = getNext(q)$  is a leaf node, one can output all solutions that use  $h_{q_N}$ . This can be achieved by maintaining one stack per node in the query.

[0075] When given a query twig pattern  $q$  and an XML database  $D$ , Algorithm TwigStack correctly returns all answers for  $q$  on  $D$ . Consider a query twig pattern  $q$  with  $n$  nodes, and only ancestor-descendant edges, and an XML database  $D$ . Algorithm TwigStack has worst-case I/O and CPU time complexities linear in the sum of sizes of the  $n$  input lists and the output list. Further, the worst-case space complexity of Algorithm TwigStack is the minimum of (i) the sum of sizes of the  $n$  input lists, and (ii)  $n$  times the maximum length of a root-to-leaf path in  $D$ . Note that for the case of query twigs with ancestor-descendant edges, the worst-case time complexity of Algorithm TwigStack is independent of the sizes of solutions to any root-to-leaf path of the twig.

[0076] It is understood that the above is true only for query twigs with ancestor-descendant edges. In the case where the twig pattern contains a parent-child edge between two elements, Algorithm TwigStack is no longer guaranteed to be I/O and CPU optimal. In particular, the algorithm might produce a solution for one root-to-leaf path that does not match with any solution in another root-to-leaf path.

[0077] Consider the query twig pattern with three nodes:  $A$ ,  $B$  and  $C$ , and parent-child edges between  $(A, B)$  and between  $(A, C)$ . Let the XML data tree consist of node  $A_1$ , with children (in order)  $A_2, B_2, C_2$ , such that  $A_2$  has children  $B_1, C_1$ . The three streams  $T_A, T_B$  and,  $T_C$  have as their first elements  $A_1, B_1$  and  $C_1$ , respectively. In this case, one cannot say if any of them participates in a solution without advancing other streams, and one cannot advance any stream before knowing if it participates in a solution. As a result optimality cannot be guaranteed.

[0078] Algorithms PathStack and TwigStack process each node in the input lists to check whether or not it is part of an answer to the query (path or twig) pattern. When the input lists are very long, this may take a significant amount of time. As described below, a variant of B-trees, denoted XB-tree, can be used on the input lists to speed up processing.

[0079] The XB-tree is a variant of the B-tree designed for indexing the positional

representation (DocId, LeftPos : RightPos, LevelNum) of elements in the XML tree. The index structure when all nodes belong to the same XML document is described below; the extension to multiple documents is straightforward.

[0080] The nodes in the leaf pages of the XB-tree are sorted by their LeftPos (L) values, which is similar to the leaf pages of a B-tree on the L values. The difference between a B-tree and an XB-tree is in the data maintained at internal pages. Each node N is an internal page of the XB-tree consisting of a bounding segment [N.L, N.R] (where L denotes LeftPos and R denotes RightPos) and a pointer to its child page N.page (which contains nodes with bounding segments completely included in [N.L, N.R]). The bounding segments of nodes in internal pages might partially overlap, but their L positions are in increasing order. Besides, each page P has a pointer to the parent page P.parent and the integer P.parentIndex, which is the index of the node in P.parent that points back to P. The construction and maintenance of an XB-tree is similar to that of a B-tree, using the L value as the key; the difference is that the R values need to be propagated up the index structure.

[0081] Using an XB-tree, a pointer  $act = (actPage, actIndex)$  to the  $actIndex$ 'th the node in page  $actPage$  of the XB-tree is maintained. Two operations over the XB-tree that affect this pointer include advance and drillDown. For operation advance, if  $act = (actPage, actIndex)$  does not point to the last node in the current page, one simply advances  $actIndex$ . Otherwise,  $act$  is replaced with the value  $(actPage.parent, actPage.parentIndex)$  and recursively advances it.

[0082] For operation drilldown, if  $act = (actPage, actIndex)$ ,  $actPage$  is not a leaf page, and N is the  $actIndex$ 'th node in  $actPage$ ,  $act$  is replaced with  $(N.page, 0)$  so that it points to the first node in N.p.

[0083] Initially  $act = (rootPage, 0)$ , pointing to the first node in the root page of the XB-tree. When  $act$  points to the last node in  $rootPage$  and it is advanced, the traversal is

finished. Algorithm TwigStackXB, shown in FIG. 9, extends Algorithm TwigStack so that it uses XB-trees. The only changes are in the lines indicated by parentheses. The function isPlainValue returns true if the actual pointer in the XB-tree is pointing to a leaf node (actual value in the original stream). If one defines isPlainValue(T)=true when T is not an XB-tree but a regular file, this algorithm reduces to the previous one. Given a query twig pattern  $q$  and an XML database  $D$ , Algorithm TwigStackXB correctly returns all answers for  $q$  on  $D$ .

[0084] FIG. 9A shows an exemplary sequence of steps for implementing TwigStackXB. It is understood that there is overlap between processing steps for TwigStack (FIG. 8A) and TwigStackXB (FIG. 9A), which includes indexing. Accordingly, to avoid unnecessary redundancy of description steps with significant commonality will have the same reference number with the addition of a prime, i.e., “'” in FIG. 9A. In step 200' it is determined whether the stream  $q$  is not empty. If so, in step 300 the node  $qact$  is retrieved and in step 302 it is determined whether the node  $qact$  is an index leaf node. If so, then processing continues with step 204 ' etc. If not, then in step 304, it is determined whether node  $qact$  is part of a solution. If the node  $qact$  is not part of the solution then in step 212' the stream containing the node  $qact$  is advanced through the index and processing continues in step 200'. If the node  $qact$  is part of the solution in step 306 the index is descended and processing continues in step 200'.

[0085] Experimental results on the efficiency of Algorithm TwigStackXB described below show that it performs matching of query twig patterns in sub-linear time. The inventive XML join algorithms were implemented in C++ using the file system as the storage engine. Experiments were run on a 550Mhz Pentium III processor with 768MB of main memory and a 2GB quota of disk space. Synthetic and real-world data were used. The synthetic data sets are random trees generated using three parameters: depth, fan-out and number of different labels. For most of the experiments presented involving synthetic data sets, full binary and ternary trees were generated. Unless specified explicitly, the node labels in the trees were uniformly distributed. Other configurations

(larger fanout and random depths in the tree) were tried including the use of the so-called XMach-1, and XMark benchmarks.

[0086] The real data set is an “unfolded” fragment of the DBLP database. In the DBLP dataset, each author is represented by a name, a homepage, and a list of papers. In turn, each paper contains a title, the conference where it was published, and a list of coauthors. The unfolded fragments of DBLP were generated as follows. It was started with an arbitrary author and converting the corresponding information to XML format. For each paper, each coauthor name was replaced with the actual information for that author. The Unfolding of authors was continued until reaching a previously traversed author, or a depth of 200 authors. The resulting XML data set has depth 805 and around 3 million nodes, representing 93,536 different papers from 36,900 unique authors.

[0087] In the experiment described below, the inventive holistic PathStack algorithm was compared against strategies that use a combination of binary structural joins. For this purpose, a synthetic data set was used consisting of 1,000,000 nodes and six different labels:  $A_1, A_2, \dots, A_6$ . Note that the actual XML data can contain many more labels, but that does not affect the techniques since one only access the indexes of labels present in the query. The path query  $A_1//A_2//\dots//A_6$  was issued and evaluated using PathStack. Then, all binary join strategies resulting from applying all possible join orders were evaluated. FIG. 10 shows the execution time of all join strategies, where each strategy is represented with a bar. There is also shown with a solid line the execution time of PathStack, and with a dotted line the time it takes to do a sequential scan over the input data (labeled SS).

[0088] For this query, the PathStack algorithm took 2.53 s, slightly more than the 1.87 s taken by the sequential scan over the input data. In contrast, the strategies based on binary structural joins ranged from 16.1 s to 53.07s. One conclusion is that optimization plays a role for binary structural joins, since a bad join ordering can result in a plan that is more than three times worse than the best plan. Another conclusion is that the holistic

strategy is superior to the approach of using binary structural join for arbitrary join orders. In this example, it results in more than a six-fold improvement in execution time over the best strategy that uses binary structural joins.

[0089] The efficiency of the different holistic path join algorithms described above can be evaluated. For example, the two versions of PathMPMJ can be compared. A 64k synthetic data set can be used, with labels  $A_1, \dots, A_{10}$ , and issue path queries of different lengths. FIG. 11 shows the execution times of both techniques, as well as the time taken for a sequential scan over the input data. Algorithm PathMPMJNaive is slower compared to the optimized PathMPMJ (generally over an order of magnitude). It appears that PathMPMJNaive is overly conservative when backtracking and reads several times unnecessary portions of the data (in one experiment, as much as 15 times more nodes than PathMPMJ). Since the performance of PathMPMJNaive degrades considerably with the size of the data set and the length of the input query, this strategy is not considered further.

[0090] Algorithm PathStack is now compared against PathMPMJ. FIGs. 12A and 12B show the execution time and the number of nodes read from disk for path queries of different lengths and a synthetic data set of 1,000,000 nodes and 10 different labels. It can be seen that algorithm PathStack provides considerably better performance than PathMPMJ, and this difference increases with longer path queries. This appears to be explained by the fact that PathStack makes a single pass over the input data, while PathMPMJ needs to backtrack and read again large portions of data. For instance, for a path query of length 10, PathMPMJ reads the equivalent of five times the size of the original data, as seen in FIG. 12B. In FIG. 12A, for path queries of length two, the execution time of PathStack is considerably slower than that of the sequential scan, and closer to PathMPMJ. This behavior is due to the fact that for the path query of length two, the number of solutions is rather large (more than 100,000), so most of the execution time is used in processing these solutions and writing them back to disk. For longer path queries, the number of solutions is considerably smaller, and the execution of PathStack

is closer to a sequential scan and more efficient than PathMPMJ.

[0091] FIG. 13 shows the execution time and number of values read for two simple path queries over the unfolded DBLP data set (note the logarithmic scale on the Y axis). Due to time-specific nesting properties between nodes in this data set, the PathMPMJ algorithm spends much time backtracking and reads several times the same values. For instance, for the path query of length three in FIG. 13, PathMPMJ reads two orders of magnitude more elements than PathStack.

[0092] Now examining twig queries, TwigStack can be compared against the native application of PathStack to each branch in the tree followed by a merge step. As described above, TwigStack is optimal for ancestor/descendant relationships, but it may be suboptimal for parent/child relationships.

[0093] FIGs. 14A-C show a series of twig queries. The twig query of FIG. 14A was used over different synthetically generated data sets. Each data set was generated as a full ternary tree. The first subtree of the root node contained only nodes labeled  $A_1, A_2, A_3$  and  $A_4$ . The second subtree contained nodes labeled  $A_1, A_5, A_6$  and  $A_7$ . Finally, the third subtree contained all possible nodes. Thus, there are many partial solutions in the first two subtrees but those do not produce any complete solution. Only the third subtree contains actual solutions.

[0094] The size of the third subtree was varied relative to the sizes of the first two subtrees from 8% to 24% (beyond that point the number of solutions became too large). FIGs. 15A-B show the execution time of PathStack and TwigStack and the number of partial solutions each algorithm produces before the merging step. The consistent gap between TwigStack and PathStack results from the latter generating all partial solutions from the first two subtrees, which are later discarded in the merge step ( $A_1//A_2//A_3//A_4 \subseteq (A_1//A_5//A_6//A_7)$ ). As can be seen in FIG. 15B, the number of partial solutions produced by PathStack is several orders of magnitude larger than that of the TwigStack

algorithm. The number of solutions to the query computed by both algorithms is, of course, the same.

[0095] The twig query of FIG. 14B was then used. Different synthetic data sets were generated in the following way. As before, each data set is a full ternary tree. The first subtree does not contain any nodes labeled  $A_2$  or  $A_3$ . The second subtree does not contain any  $A_4$  or  $A_5$  nodes. Finally, the third subtree does not contain any  $A_6$  or  $A_7$  nodes. Therefore, there is not even a single solution for the query twig, although each subtree contains a large number of partial solutions. One difference with the previous experiment is that one needs to materialize an intermediate join result before getting the final answer. Therefore, there is no execution strategy using PathStack that avoids materializing a relatively large intermediate result.

[0096] FIG. 15C shows the execution time for PathStack and TwigStack for different data sizes (note the logarithmic scale). For the last data set (with 243K nodes), PathStack could not finish since the intermediate result filled all the available space on disk (2GB).

[0097] As discussed above, TwigStack is not optimal for parent/child relationships. Even in this case, TwigStack performs better than PathStack. The queries in FIGs. 14A and 14B were modified by adding the following constraint: all ancestor-descendant relationships are connected by a path of length between one and three (this can be checked by comparing the LevelNum values of the positional representations).

[0098] FIGs. 16A-C show the results for these experiments. Even in the presence of parent-child constraints, TwigStack is considerably more efficient than PathStack. In particular, FIG. 16B shows that the number of partial solutions produced by TwigStack (though not minimal) is small. The non-minimality is evident from the observation that the number of partial solutions produced by TwigStack is sometimes larger than the number of total solutions to the query twig.

[0099] The query of Figure 14(c) was also evaluated over the unfolded DBLP data set. This query asks for authors with papers published in the year 2000 who have some coauthor with a paper published in 1990, who in turn has some coauthor with a paper in 1980. The allowed depth was varied in the relationship COAUTHOR // PAPER, i.e., the number of coauthor and papers one can traverse from a given author, from 0 (no solutions) to 37. The results are shown in FIGs. 17A and 17B. It can be seen that for these queries, TwigStack is again more efficient than PathStack.

[0100] The advantages of using XB-trees to process path and twig queries can be evaluated. In particular, it is shown that the number of nodes that need to be read from the XB-tree (counting both leaf and internal nodes) is significantly smaller than the size of the input, which causes sub-linear behavior in the inventive algorithm. As will be seen, XB-trees with small node capacities can effectively skip many leaf nodes, but the number of internal nodes traversed is large. On the other hand, for large node capacities there are fewer internal node accesses, but XB-trees cannot skip many leaf nodes because they could miss some solutions. The best experimental results were obtained when using node capacities ranging from 4 to 64.

[0101] For these experiments, different queries were evaluated using PathStack and TwigStack, with and without XB-trees. The node capacity of the XB-trees was varied between 2 and 1,024 values per index node. FIG. 18A shows the number of values read in the XB-tree (separated into internal and leaf accesses) for the data set and path queries. FIG. 18B shows the results when using the twig query of FIG. 14A. FIG. 18C shows the results for the twig query in FIG. 14C over the unfolded DBLP data set.

[0102] In general, the total number of nodes visited in the XB-Tree is consistently smaller than the input data size for a wide range of node capacities. For the synthetic data set, better results were obtained for complex queries. In those situations, XB-Trees can prune significant portions of the input data. In contrast, for simpler queries, one needs to go deep in the XB-Tree nodes, in many cases down the leaves, since there are many